СРЕДА ПРОГРАММИРОВАНИЯ, СТРУКТУРИРОВАННАЯ В ТЕРМИНАХ ЯЗЫКА

Черясов Д. Г,
Самарский государственный аэрокосмический университет, г. Самара

Предлагается нетекстовая среда программирования, не имеющая большинства недостатков традиционной текстовой формы представления программ. Полностью устраняется возможность появления языковых ошибок. Возможны различные визуальные представления и способы редактирования программ. Существенно упрощается трансляция; среда создания программ легко интегрируется со средой исполнения. Все указанное достигается использованием специального вида исходной формы программы, в которой программа хранится в структурированном (разобранном) виде.

1. Трудности текстового представления программ

Традиционной формой представления программы является "исходный текст", рассматриваемый большинством трансляторов как поток символов или строк. Текст сам по себе не структурирован в терминах, описывающих программу, поэтому требуется использование текстовых конструкций, по которым транслятор выявляет структуру (ключевые слова, символы и пр.). Для описания сложных вложенных структур применяются парные конструкции (скобки и т. п.). Простой текстовый редактор позволяет легко нарушить подобные структуры. Большинство языков накладывает на программу семантические ограничения, такие, как использование пространств имен, соответствие типов в выражениях или требование декларировать объект до использования. Этот аспект не может быть проконтролирован без трансляции в процессе редактирования. В результате изменение исходного текста программы обычно ведет к появлению ошибок, обнаруживаемых лишь при трансляции. Такие ошибки мы будем называть *"языковыми"*.

Всякая запись программы, с которой имеет дело человек, должна быть способна содержать некоторую метаинформацию (комментарии и т. п.). Текстовая форма предоставляет только один тип метаинформации — "комментарий". Контекст, в котором рассматривается комментарий, не определен (где границы фрагмента кода, к которому относится комментарий `/* ошибка здесь */` ?) Метаинформация различного рода, например, для связи с построителями графического интерфейса пользователя (ГИП) представляется через "комментарии специального вида", которые легко случайно нарушить.

Традиционный инструмент для написания программ — текстовый редактор — не способен редактировать программу непосредственно в терминах языка, поскольку текст как форма представления не обладает адекватной структурой. По этой же причине затруднено использование нетекстовых инструментов редактирования (таких, как графические редакторы иерархии классов, структуры проектов и т. п.), предполагающих некоторый предварительный разбор текста и привязку к нему.

Транслятор должен разбирать исходный текст, то есть декодировать то, что программист вынужден был закодировать в виде текстовых конструкций. Сложность этого процесса общеизвестна [1]. Это мешает, например, транслировать только измененные части программы, поскольку обычно невозможно априори сказать, какие части текста с какими структурами программы связаны.

2. Представление программы, структурированное в терминах языка

Понятие *исходной формы программы* (ИФП) расширяет понятие исходного текста программы на частично или полностью нетекстовые представления. Покажем, как можно построить ИФП, лишенную многих недостатков текстовой ИФП: такую, в которой в принципе невозможны языковые ошибки, которая позволяет представлять программу различными способами, хранить структурированную метаинформацию и упростить трансляцию. Построим ИФП, структурированную в терминах целевого языка программирования высокого уровня. Элементами ИФП в таком случае станут семантически элементарные конструкции языка (например, введенные в [2]), а саму ИФП будет удобно представляться в виде "дерева разбора" или подобной структуры.

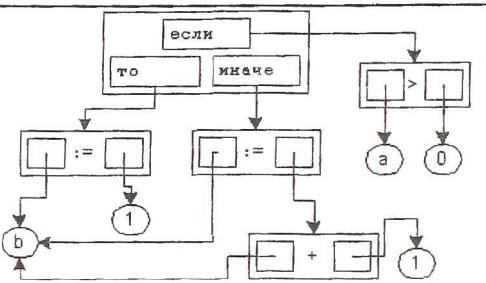
Рассмотренная далее ИФП оперирует с (семантически значимыми) элементами языка как с *неделимыми сущностями*. Такой подход, вообще говоря, применим к очень широкому кругу языков, например, к языкам разметки текста типа SGML (HTML, XML).

2.1. Языковые объекты как основа ИФП

Сопоставим каждой конструкции языка, удобной с точки зрения семантики, некоторый неделимый *языковый объект* (ЯО). Каждый ЯО может иметь *слоты*, в которые помещаются ЯО, представляющие вложенные конструкции. ЯО являются единственными строительными блоками нашей ИФП; дерево языковых объектов соответствует дереву

разбора программы, чем поддерживается языковая целостность программы в процессе ее редактирования. (Обобщенный подход описан, например, в [3].) ЯО, не имеющие слотов (то есть вложенных частей), назовем *простыми*; очевидно, наше дерево разбора будет иметь в качестве листьев только их. Следующий пример иллюстрирует описанный подход.

Фрагмент текста программы	Структура ЯО, представляющая фрагмент слева

`if (a>0)`
`then b:=1`
`else b:=b+1;`
	

Внешние прямоугольники представляют сложные ЯО, внутренние прямоугольники со стрелками представляют слоты; круглую форму имеют простые ЯО. Видно, что в обоих присваиваниях и сложении используется та же самая переменная b.

Языковые объекты обладают следующими свойствами:

1. Любой ЯО создается и уничтожается как единое целое;
2. Каждый ЯО имеет слоты для вложенных конструкций, если таковые допустимы. (*Пример.* ЯО для оператора ветвления `if` имеет три слота: для условия, для блока `then` и для блока `else`);
3. Каждый ЯО заботится о поддержании собственной корректности; для этого он проверяет состояние объемлющих и вложенных ЯО. Содержимое слотов любого ЯО корректно с момента его создания;
4. ЯО не допускает некорректных операций с собой;
5. Начальный (минимальный корректный) "текст" состоит из некоторого ЯО верхнего уровня, такого, как модуль в Modula-2 или package в Java. Слоты каждого вновь созданного ЯО заполняются некоторыми значениями по умолчанию.

Построение программы из таких ЯО исключает любые языковые ошибки; для этого на стадии редактирования производится постоянная проверка корректности. Высокая скорость проверки достижима за счет того, что элементарная операция редактирования обычно вносит небольшие изменения в ограниченном контексте, поэтому объем

анализируемой информации мал. ЯО содержат взаимные ссылки, позволяющие быстро определять их взаимозависимость и ограничить контекст, в котором необходима проверка. В большинстве случаев проверка корректности дерева ЯО тривиальна.

Для помещения в ИФП метаинформации имеется особый класс ЯО, подклассами которого являются различные формы комментариев. Для метаинформации имеется специальный слот в каждом ЯО; метаинформацию можно помещать только в эти специальные слоты, чем достигается точное определение ее контекста.

Можно расширять язык, создавая специализированные языковые объекты, например, для параллельных вычислений, при этом не теряя возможности комбинировать их с уже существующими. Это позволяет создавать диалекты, ориентированные на специфическую предметную область.

2.2 Слот как связующий элемент ИФП

Слот — это такой компонент ЯО, в который всегда помещены другие ЯО, удовлетворяющие некоторым ограничениям. Иначе говоря, слот соответствует (некоторым) дугам между узлом и его подузлами в дереве разбора. (Сравните с [4].) С точки зрения традиционной грамматики, слот соответствует вхождению одного нетерминала в определение другого нетерминала: например, правило

<цикл> ::= "пока" <лог-выраж> {<оператор>} "все"

содержит 2 слота: слот для логического выражения (инварианта цикла) и слот для операторов цикла.

Представим ЯО как кортеж <c, S>, где c — некоторый "*класс*" ЯО, S — упорядоченное множество слотов, принадлежащих ЯО. В нашем примере ЯО <цикл> соответствует кортеж <"оператор", (инвариант, тело)>.

Слот представим как кортеж <c, ft>, D, где c — класс допустимых в слоте ЯО, ft(O) — функция проверки, истинная тогда и только тогда, когда языковый объект O можно поместить в слот, D — упорядоченное множество ЯО, помещенных в данный слот.

(Обозначая элемент e кортежа C, будем писать C.e.)

Классы слотов и ЯО понимаются как классы в смысле ООП и составляют одну иерархию. Для классов введем следующие операторы:

- (тождество) c1 = c2 истинно тогда и только тогда, когда c1 и c2 — один и тот же класс;

- (совместимость) c1 > c2 истинно тогда и только тогда, когда c1 = c2 или c1 является подклассом c2.
- (принадлежность) O **is** c истинно тогда и только тогда, когда O есть синтаксический объект класса c или O **is** c1 и c1 c.

Поэтому языковый объект O можно поместить в слот S, только если O.c > S.c. Иначе, not (O is S.c) => S.ft(O) = false, обратное верно не всегда, так как могут иметься дополнительные языковые ограничения.

Таким образом, именно механизм слотов поддерживает языковую корректность ИФП.

2.3. О трансляции и исполнении

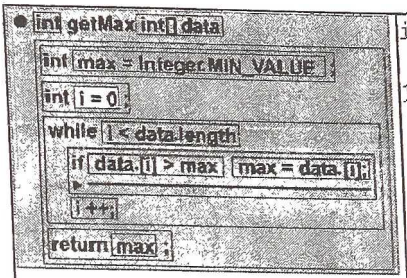
Рассмотренная ИФП фактически осуществляет то, чем для текстового представления является синтаксический разбор и семантический контроль, поэтому трансляция сводится к кодогенерации и оптимизации. В результате время трансляции можно резко снизить. Также можно заново транслировать лишь измененные участки. Можно модифицировать набор ЯО таким образом, чтобы каждый из них генерировал код для своей части программы, или же интерпретировал ее.

3. Редактор для нетекстовой ИФП

Описанная исходная форма не представляет интереса без средств редактирования, то есть средств ее визуального представления и модификации. Использование описанной ИФП позволяет сделать эти средства весьма богатыми.

3.1. Визуальное представление

Простейший вариант визуализации — сопоставить каждому ЯО некоторое изображение (например, геометрическую фигуру); изображения, соответствующие ЯО, находящимся в слотах, можно поместить внутрь изображения объемлющего ЯО.




```
int getMax(int[] data) {
 int max =
Integer.MIN_VALUE;
 int i = 0;
 while (i < data.length) {
  if (data[i] max)
   max = data[i];
  i++;
 };
 return max;
};
```

Знание структуры программы позволяет рассматривать ИФП "крупным планом", "сворачивая" некоторые ее части [5], например, можно скрыть тело функции, оставив лишь ее декларацию. Такой способ просмотра дает возможность видеть (и создавать) структуру программы в виде шагов при проектировании по методу "сверху вниз".

Можно придать ИФП свойства гипертекста: например, можно перемещаться между местом использования объекта и его декларацией или переопределением. Внесение специального класса метаинформации "закладка" позволяет создавать произвольные гиперсвязи, поскольку закладки могут ссылаться на другие закладки.

Для разных задач можно иметь несколько способов визуализации каждого ЯО (например, удобный для навигации и удобный для редактирования). Возможность переключаться между ними облегчает работу, позволяя рассматривать структуру программы наиболее удобным способом.

3.2. Редактирование и контроль корректности

Редактирование описанной ИФП можно осуществлять, например, методом буксировки. Каждый ЯО может быть отбуксирован из другой части ИФП или некоторого "образцового" набора ЯО в нужный слот в ИФП. Доступ к уже существующим в ИФП объектам также осуществляется через их список. Такой список, в котором перечислены, например, все объявленные в программе переменные, доступные в данном контексте, может быть весьма велик, поэтому разумно ввести также "горячий" список объектов, которые с наибольшей вероятностью могут понадобиться в данном контексте. (Например, поля класса — хорошие кандидаты в "горячий список" при редактировании метода класса.)

Контроль корректности формы при операциях редактирования может занимать заметное время. При редактировании методом буксировки важно уметь быстро оценить корректность действия, чтобы сообщить результат пользователю, например, изменив форму курсора.

Опишем простой вариант функции S.ft(O), допускающий регулировку точности проверки в случае ограничений по времени. Входные параметры: языковый объект O и слот S. Функция возвращает одно из трех значений: "истина", "ложь" и "возможно", третий вариант соответствует случаю, когда опровержение не найдено в связи с ограничением на глубину просмотра.

I. В случае операции добавления/перемещения проверяется O.c > S.c. В случае несовместимости возвращается "ложь".

II. Результату присваивается "истина".

III. Помечаются все ЯО, которые могут быть затронуты данным изменением.

IV. Каждый помеченный ЯО (рекурсивно) проверяется на корректность в будущем новом состоянии.

A. Если корректность не может быть достигнута без изменений, производится поиск разумного преобразования этого ЯО (см. ниже).

B. Если такое преобразование возможно, но наложен запрет на (дальнейший) рекурсивный вызов, результату присваивается "возможно", иначе для проверки корректности этого нового изменения весь данный алгоритм вызывается рекурсивно.

V. Если корректность достигается, пометка с ЯО снимается.

VI. Если остались помеченные ЯО, возвращается "ложь", иначе возвращается результат.

Параметром "запрет на рекурсивный вызов" можно регулировать максимальную глубину просмотра и, соответственно, время проверки. Можно задать фиксированную глубину просмотра или динамически ограничивать рекурсию, исходя из фактически затраченного времени. Это важно, когда нужно поддерживать визуальную обратную связь в реальном времени, например при буксировке; полная проверка в таком случае производится при отпускании буксируемого ЯО.

При каждом запросе на изменение ИФП происходит полная проверка нового состояния на корректность и, если оно корректно, изменение вносится.

Отдельную проблему представляет смена контекста, когда часть кода перемещается из одного места ИФП в другое. Из-за возможности существования локальных контекстов, сходных по структуре (например, локальных пространств имен) может потребоваться такое преобразование ЯО, чтобы ИФП осталась корректной, если это возможно. В приведенном ниже примере переменной i после перемещения указанного фрагмента текста должен соответствовать иной ЯО, соответствующий одноименной переменной в другом локальном пространстве имен:

до перемещения	после перемещения

`int foo (int i) {`
`...`
`/* фрагмент */`
`i=i+1;`
`...`
`};`
`int bar (int i)`
`{...};`
	`int foo (int i) {...};`
	`int bar (int i) {`
	`...`
	`/* фрагмент */ i=i+1;`
	`...`
	`};`

При перемещении (копировании) значительных объемов кода проблема преобразования контекста может стать достаточно сложной. Для облегчения этого процесса вводятся "разделочные доски" — специальные окна редактирования, в которые можно копировать фрагменты кода вместе с требующимся им контекстом и затем изменять код так, чтобы он соответствовал указанному целевому контексту.

Список литературы

1. Теория синтаксического анализа, трансляции и пр., 1 том. Ахо и Ульман. // М., Мир, 1978
2. Дисциплина программирования. Э. Дейкстра. // М. Радио и связь, 1982
3. Abstract Visual Syntax. Martin Erwig. // Proc. of TVL-97 (http://www.pst.informatik.uni-muenchen.de/~bmeyer/TVL97/TVL97papers/TVL97erwig.ps.gz)
4. Programming with Visual Expressions. Wayne Citrin, Richard Hall, Benjamin Zorn // Proc. of TVL-95
5. Improving Readability of Iconic Programs with Multiple View Object Representation. Yuichi Koike, Yasuyuki Maeda and Yoshiyuki Koseki. // Proc. of TVL-95.